U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 89-4053

National Computer Systems Laboratory

Architecturally-Focused Benchmarks with a Communication Example

G.E. Lyon
R.D. Snelick

U. S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Advanced Systems Division
Gaithersburg, MD 20899

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

March 1989

Partially sponsored by the
Defense Advanced Research Projects Agency.

ARCHITECTURALLY-FOCUSED BENCHMARKS WITH A COMMUNICATION EXAMPLE

G.E. Lyon and R.D. Snelick

Advanced Systems Division
National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Partially sponsored by the
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209.

U.S. Department of Commerce, Robert A. Mosbacher, Secretary

Ernest Ambler, Acting Under Secretary for Technology

National Institute of Standards and Technology
Raymond G. Kammer, Acting Director

March 1989

TABLE OF CONTENTS

ARCHITECTURALLY-FOCUSED BENCHMARKS WITH A COMMUNICATION EXAMPLE

G.E. Lyon and R.D. Snelick

The discussion first sketches a framework of modalities for an architecturally-focused performance evaluation. The result is a hybrid of benchmarking and modeling: Elements of capacity-and-use trees, CUTs, are explored as a simplified notation. There follows a description of the structure and preliminary results from a practical benchmark set for process communication.

Argument is given that performance within a class of architecture is often dominated by unavoidable competitions within distinct machine modalities, such as scalar-vector. A k-alternative, forced choice defines a dimension of comparison equally well in SIMD or MIMD architectures. Performance estimators are interpolations between values from basis benchmarks for modes; ideally in the two-alternative forced choice only two benchmark measurements are needed. Refinements in basis benchmarks support CUT-based estimates of performance.

The example set of communication benchmarks shows how refinements can clarify knowledge of a machine. The refining expands details for a given mode. Ring, mesh, and random connection benchmarks demonstrate diagnostic details on a particular mode (process communication) on a machine. The results sample both shared-memory and message-passing, and cover architecture influence, synchronization mechanisms, message, computational and synchronization granularities, and mappings of logical to physical structures. Emphasis is upon capturing important characteristics and capabilities of a machine's communication subsystem.

Key words: architecture; benchmarks; measurements; metrics; models; performance; synthetics.

The identification of commercial products in the text is for clarification of specific concepts. In no case does such identification imply recommendation or endorsement by NIST, nor does it imply the product is necessarily the best for the purpose. Partially sponsored by the Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Virginia 22209.

1. Introduction

This report serves two related purposes. One is to examine some foundations of benchmarking. The other is to demonstrate practical results. Although discussion begins with the idea of architecturally-focused machine evaluation, this is not to say that performance evaluations set against applications are not important. They are. Such has been argued earlier [LYO87, LYO88] and is reflected in the set of communication benchmarks described in the latter half of the text. Nonetheless, an economy in machine-based evaluation makes it attractive. Particular emphasis is given to competing programming modes, e.g., scalar versus vector, and to a coding-theoretic notation of this information called a capacity-and-use tree, or *CUT*. The CUT explicitly displays hardware and application assumptions via benchmark results, which decorate its branches and leaves.

A second brief topic covers a distinction between parallel *speedup* and the related term *scaleup*. It appears that *scaleup* practices may be prevalent in some fields [GUS88], although the term *speedup* is unfortunately applied somewhat indiscriminately. Scaleup is easier to achieve, provided it can be done.

The third section, half the report, provides experimental results from a communication benchmark set developed within our parallel processing project at NIST. Preliminary results cover a ring, mesh, and random organization on both shared-memory and loosely coupled systems. Focus is upon process-level transactions. This section illustrates using benchmark refinements within a given mode to discover strengths and weaknesses.

2. Architecturally-Focused Performance Evaluation

Computers are notoriously elusive to characterize. For performance, characterizations are usually torn between two camps of thought. The first provides a large battery of comprehensive tests that reflects the application world [VAN88]. Unfortunately, these tests may themselves be hard to characterize, and the ensemble expensive to administer: applications are exceedingly diverse. A second view, taken here, tries to focus only upon the barest architectural features. The hope is that this economy, although providing rougher measurements, will provide a broader, more accessible summary of salient facts. While it is generally agreed that the first approach, with its knowledge of applications, is the fullest characterization [POT87], there are often good reasons why an architecturally-focused approach is better suited. A large number of machine choices may need preliminary winnowing. Or the application community may be poorly defined, as with machines whose export a government wants to control. In export, only the machine architecture and operating system are definitely known [BAI88].

2.0.1 Assumptions on Comparability. The nature of the restricted software viewpoint should be explained. While an architectural focus might seem to preclude applications, there are necessary exceptions. *Operations* are defined *logically* at the language level, as in FORTRAN, Pascal, or Ada®. However, this is not to say that textual reoccurrences of language expressions are invariably computed the same way on a machine. Indeed, a principal tenet is that such is often not the case.

Any machines to be compared run essentially the same logical instructions, even though respective machine codes are distinct. It is assumed that algorithmic changes are not needed because machines in a class are adequately comparable. This assumption is used later to simplify a hybrid model into a simple, tabular format.

2.1 Architecture and Benchmark Evaluations

Ideally, one would like to simplify specifications and to provide major performance characteristics of a machine. The point is related to performance modeling, and entails many of the same hazards. For example, it is important that emphasized features dominate performance. General questions on a machine's fundamental balance and capabilities include:

* size of memories
* processor bandwidths
* i/o capabilities
* memory-to-processor bandwidths
* processor-to-processor communication
* memory move-around (memory-to-memory bandwidth)

Hillis claims, with good justification, that such coarse characteristics must be in reasonable balance and accord to warrant serious attention [HIL85]. As an example, a machine with a 2 MIP (million-instruction/sec) processor should not have but a 4 kilobyte memory--such a memory would be too small to complement capabilities of the processor. The above list is a good minimal tally, a place to start.

The focus is upon benchmarks. This is in contrast to any attempt at modeling via simulation or analysis. The very fine-detailed complexities of modern machines can render modeling an ambitious undertaking. Something simpler is desired. Determining important machine aspects for benchmarking may be a straightforward interpretation from the architecture, but this is not guaranteed. Even the above list can be interpreted as more of a general functional requirement of applications than an imperative of what a machine must have. Machines hold surprises in capabilities established not through obvious architectural features, but through synergistic strengths and weaknesses of component groups acting against demands. These strengths and weaknesses exert a "hidden topology" of performance that must be discovered. One pivotal "topology" element is the mode of operations. The essentials of this emphasis were described by G. Amdahl [AMD67] and recast by Ware [WAR72]. However, operation modes are merely a departure point. Focus is at a logical, rather than machine level.

2.1.1 Modes and Operations. Operations can be classified as *reduced* or *multimodal*. An operation's designation depends upon architecture and implementation. For this reason, special benchmarking attention must be given to catch modal details:

> *Reduced operations.* The term *reduced* denotes an operation that behaves more or less the same under a variety of processor- and system-state circumstances. Operations are single-

mode. On older machines, operations were often quite predictable. Formulae were even provided by manufacturers to calculate the clock cycles for each instruction. In the terminology, these operations were *reduced*, or primitive.

Multimodal operations. The modern machine may have a complex behavior in terms of timings for a single logical operation. Examples include: mapping by compilers to scalar or vector instructions; non-linear operation speeds that hinge upon the size of instruction context (instruction cache); memory-fetch anisotropy generated by incongruities between virtual memory and actual physical allocation.

2.1.2 Multimodal Competition. Operation modes compete against each other during program execution. Competing alternatives do not provide the same rate of computation, but *each mode is useful enough that it cannot be ignored.* For example, scalar operations have rapid starts. All vector supercomputers have operating systems, and these operating systems run in *scalar* mode. Ten to fifteen percent of a machine's instruction dispatches support operating system services. Beyond scalar instructions of the operating system, many practical application codes of strategic interest need scalar support. These codes include Monte Carlo photon transport, tri-diagonal elimination, table lookups and interpolations for equation of state; this group may vectorize *not at all.* Particle-in-cell and banded linear equations may vectorize for only about half the executed operations. So scalar capabilities are important, even if they are slower for linear equation systems, which vectorize well. The early Star-100 vector machine was soon replaced by the 203/205 series. The Star had good vector processing but inadequate scalar capabilities; the 203 had improved scalar speeds [HOC84].

A major difficulty with multimodal operations is that they can appear *ad hoc*, varying from implementation to implementation (even without architectural changes). Caches, compiler quality, memory types, and other low-level details have effects. To handle the variability that complex enhancers of performance introduce, one often tries to sample likely parameters of a language-level operation to establish a range of performance. This is discussed shortly as a refinement of the context of some basis measurements. Furthermore, the definition of multimodal operation is not meant to be strict in its wording. As case in point, the competitors of (i) unit vector stride, versus, (ii) some random fetching combined with GATHER/SCATTER, do fall into the multimodal category. From an operator viewpoint, it is whether to use GATHER/SCATTER, but stride actually determines the choice. Stride is the independent variable.

A modality must form a *k-alternative, forced choice of modes.* It is on the basis of the forced choice (no multiple selections) that a clear-cut dimension of performance variation emerges. Furthermore, on some architectures, one modality conflict *dominates* all others. This is certainly true for scientific computations on machines with scalar and vector capabilities.

Example 1: A dominant competition. A classical competition occurs on machines that perform either scalar or vector computations. Answers are the same done either way, but since vector processing rates (examples in [DON88]) are four to twenty-five times higher, the vector mode is naturally preferred. Although naive users of vector machines may evaluate their machine solely on peak vector performance, such is a very incomplete perspective. As just argued, scalar modes do persist. Programs are mixes of vector and scalar, the balance depending upon application algorithms. To account for this, it is very common [WAR72, BUC84] to (i) estimate scalar and vector rates s, v via benchmark measurements, and (ii) derive a composite performance estimator $p_{s,v}$ from

$$1/p_{s,v}(\alpha) = \alpha/s + (1-\alpha)/v, \text{ where } \alpha=\text{"scalar" fraction}$$

The interpolation rule that $p_{s,v}(\alpha)$ exemplifies is often referred to as Amdahl's law. Additional terms can be added on the right-hand side for further competitors. The only restriction is that right-hand numerators must sum to unity. Parameter values *s*, *v* are regarded as "basis" capabilities of pure scalar and vector modes.

2.1.3 Comparing Vector Machines. A little algebra shows that the pair (s,v) and $p_{s,v}(\alpha)$ define a well-behaved expression of performance. A new vector machine *m'* can be compared, *consistently*, with another, *m*. Since the comparisons are essentially the same for *s* or *v*, argument is given only in terms of varying *s*. If both $s' \leq s$ *and* $v' \leq v$, then $p_{s',v'}$ is everywhere for α less than or equal to $p_{s,v}$ of the original machine. Similarly, if $s' > s$ *and* $v' \leq v$, then the performance of the newer machine must for some α exceed that of the first. In short, a machine is faster for some value of α if and only if at least one of its two processing rates (s' or v') is higher.

2.1.4 Application Space Parameterized. An estimator such as $p_{s,v}(\alpha)$ serves a second important purpose beyond comparisons. It characterizes performance without having to qualify the results *by specific applications or algorithms*. Because there are but two competing modes, the single parameter α compactly summarizes the combinatoric explosion of variety seen among *all* application codes *relative to the two basis benchmarks*. (For convenience, only two-way competitions are mentioned.)

The approach assumes characteristics at the (logical) operation level; one counts operations *of interest* in their various competitive modes within the benchmark codes. This can be tedious, but it must be done carefully. A central problem with interpolation between competing factors is that actual basis measurements s^* and v^* are unlikely both to characterize a pure mode ($v=v^*$ *AND* $s=s^*$), for reasons of coding limitation. Of course, this very problem is why one mode cannot eliminate another. Consequently, some residual influence of the other mode will often remain. Thus the determination of (for example) *s* and *v* is unlikely to yield the parameters without some simple algebra. Measurement v^* may in fact be for 97% vector and 3% scalar. *v* must be derived. A pure scalar measurement is easier to obtain directly. There will be two measurements and two mix ratios; *s* and *v* are fully determined.

Success of the technique depends upon generalities that the basis benchmarks may or may not successfully incorporate. Each of them has implicit assumptions; some benchmarks will have ruinous restrictions. If constraints are too great, performance measurements will not have wide enough applicability for a general estimate of machine capabilities. Yet, whenever machines can be usefully compared via their respective scalar-vector mixes (or other modes), an enormous, clarifying simplification has been achieved.

But what happens if the mix ratios are not so easily determined? Take as an example the ratio of cache-hits, which is not readily accessible at the programming level nor comparable across machines. Comparing two different machines may require a knowledge of two distinct cache-hit values, one chosen (independent variable) and the other estimated. Furthermore, caching benchmarks *c1* and *c2* may give a "high caching" and a "lower caching" measurement, but the exact separation between their two points of measurement is unknown. Keeping a programming frame of reference allows one to use an interpolation rule via Amdahl, although accuracy is now rather questionable. Of course, the more separation between measurement points, the better. However, ascertaining clearly that c1 and c2 are well separated seems to require hardware measurement [CAR88].

2.1.5 Optimistic Estimations. Amdahl's law is optimistic. That is, it assumes a perfect, loss-less mixing of modes. This may not be true because of context switchings, message latencies, vector startups, and other practical details. The importance of this is a function of what error can be tolerated, and at what benchmarking cost. The proposed approach is to avoid higher cost through "basis" benchmarks and interpolation.

Example 2: Communication versus memory. Some machines have a communication-memory competition very analogous to the scalar-vector tradeoff. A parallel processor will have a choice of (i) using its own memory, or (ii) getting a result from neighbor(s), so that $1/p_{msg,mem} = \beta/msg + (1-\beta)/mem$, β="message" fraction. This tradeoff is especially pronounced in array processors [RED88, REE88] and in other private-memory architectures, such as hypercubes. Messages from neighboring processors may logically work the same [LAU78] as private memory references, but messages are several decimal magnitudes slower. A programmer keeps interprocessor communications to a minimum, but parallel computation forces some node-to-node exchange.

The selection between messages or memory relates to (i) costs (that a programmer would like to know) for logically equivalent constructs, and (ii) the latitude that node memory size and algorithm provide [KUN85]. One might argue that system architecture should show such a distinction, but in fact the architecture *per se* does not determine the importance. After all, message-passing is "transparent", that is, by-value and very much like private memory references. It is the large distortion in performance (rather than logical function) that makes distinctions worthwhile.

2.2 Application

Scalar/vector and message/memory are not the only tradeoff pairs. Unfortunately, even the same *hardware instruction* may be played off against itself, depending upon use. Lubeck [LUB88] remarks that, "..Amdahl's Law applies to high and low scalar performance just as it does to vector/scalar integration." Modern RISC (Reduced Instruction Set Computers) are especially sensitive to the relative sizes of benchmark and application kernels that they execute. These architectures require a particularly watchful evaluation. There are several questions pivotal to the identification of modes within a modality.

1. Among the operation functionalities of a system, which are interchangeable in some respect? A knowledge of the system may be needed beyond hardware capabilities. Parametric variations of some important operation during benchmarking may reveal multiple modes. A related question asks whether the variation is important for purposes of the evaluation. Perhaps maximum-performance settings of parameters are adequate. (This is true of export evaluation.)
2. Which possibilities of substitution are admitted by algorithms? It does little good to compare tradeoffs that have no utility. Observe that this second problem was already resolved for vectorizability, since examples are known which admit various values of α. Nonetheless, it is of no use to find sterile tradeoffs that are never made. To this limited extent, algorithms and the applications world do again intrude.

3. A modality's alternatives (modes) must present distinctions in performance. Otherwise there is no point in studying them. The scalar performance mentioned by Lubeck is one example of distinctions that are somewhat subtle. Another is virtual memory, which can be quite sensitive to reference schemes, e.g. *by-row* or *by-column* for matrices.

The following table summarizes some common binary competitions, the first three of which may occur on the same machine.

	Competitive Modes	Architectural Focus	Appx. Hardware Differences	Improvement w/ Prudent Use
1	scalar vs. vector	*vector machines*	*peak vector is 4x to 25x faster*	*Monte Carlo--none lin. alg.--near peak [DON88]*
2	random GATHER or SCATTER vs. unit-stride	*memory-to-memory vector operations*	*unit-stride is 2.5x faster (at least)*	*3x to 7x estimated [88NIST], [LUB88]*
3	by-row vs. by-column FORTRAN	*virt. memory subsystem, scalar operations*	*page faults slower by 10^4; source: row refs.*	*columns--30% faster for linear eq. solver w/FORTRAN [MOL72]*
4	messages vs. memory refs.	*loosely-coupled nodes*	*memory refs--50x to 10^3x faster*	*array processor w/mesh: 10x faster [RED88]*

Four Modalities and Performance Effects

2.3 CUT: Capacity-and-Use Tree

Numerous suggestions have been made on using competing modes to estimate performance [WAR72, BUC84, LUB88]. While the usual presentation of the idea is often bimodal (just shown), trimodalities--using parallel, vector, and serial benchmarks--have been sketched [BUC84]. These first-order methods can be refined through multi-level selections. For example, rather than just a single level vector, scalar or parallel choice, let the scalar selection have a further refinement of, say, by-row or by column (as in the previous table). Although formulae are easily derived for such cascaded selections, it is more insightful to borrow a notation from coding theory. A (doubly) weighted tree--denoted a capacity-and-use tree, *CUT*--serves two purposes. It visually displays all crucial assumptions in compact, quickly surveyed format, and it supports performance estimates, which are computed off its leaf values of benchmark results. The estimators being considered are not highly accurate methods, but rather, improvements over otherwise scattered tables and misleading single figures. An example will be given. (See Figure 1.)

The maximum capacity of any CUT arc is assumed to be unity or less. Capacity along any fanout arc is relative to the capacity of the origination node, which is taken as unity. Each stage can only diminish capacity, never improve it; the CUT must be constructed to be consistent on this.

Although the CUT is a code-tree-like structure, it is used for its own purposes. The placement of modalities, such as vector-scalar, is more a function of which factors are to be examined with parameter variations, how much dominance the modality exerts, and any dependencies. In the example, the vector-scalar modality is the root fan-out because it truly dominates everything. To begin the tree with another factor would mean that numerous duplicate vector-scalar fan-outs would have to be introduced into the tree. Indeed the secondary fan-outs are each dependent modes, but this will not alway be true. Some modalities will be independent of each other. However, a node in the tree can be constructed to express conditional circumstances. In this manner, certain important compound events such as, *e.g.*, "vector preceded by scalar," can be represented. A CUT advantage is that such detail only appears as necessary to render an accurate model.

Assume from the preceding table a hypothetical vector, memory-to-memory System XXX with benchmarks such that:

1. scalar to vector rates (peak) of 0.1 to 1
2. scalar frequency of operations, 0.3
3. scalar by-row rate of 0.7 that of by-column, 1
4. by-row or by-column frequencies of 0.5 of scalar
5. GATHER-SCATTER vector rate of 0.3 that of unit-stride, 1
6. unit-stride or SCATTER-GATHER mode equally frequent at 0.5 (of vector).

XXX at its CUT root (Figure 1) has a peak efficiency of 1. Capacity admittances on arcs show fractions of efficiency preserved on the arcs. The frequency weights express average utilizations.

Applying Amdahl's law to leaves of the CUT (the frequencies sum to one) yields a true efficiency of 0.194 *relative to the application.* Each leaf of the CUT has a fraction-of-code weight and a rate associated with it; dividing the code fraction by the rate gives a contributed time. The sum of these times, inverted, yields a coefficient of efficiency, C_{eff}, that is relative to the root (peak) rate of unity. See Figure 1 for the example C_{eff} of 0.194, as indicated in the drawing. This performance corresponds with everyday experience, which seldom approaches anywhere near peak vector performance. Admittedly, the model ignores startup delays and other real-life elements, although corrections can added, either in tree arcs, or in the estimator that takes leaf values as arguments. But as a "back-of-envelope" calculation, the approach is not completely unrealistic, since variances in everyday benchmarking can be high. It is not uncommon to have a 30 to 40 percent spread from uncontrolled variables. Consequently, it is difficult to become too concerned about minor uncertainties in the CUT model. Furthermore, the CUT declares explicitly all assumptions in the estimate. This encourages economy and candor in comparisons. Incidentally, there is no reason for the tree to be balanced in depth, although the example is. What does matter is that the frequency weights on leaves sum to unity.

As a practical issue, a very complex CUT is probably not done in the spirit of the method, which is meant to be coarse, quick and explicit. Also, as CUT arborescences multiply, the demand upon application specification grows. Each fan-out in the tree needs more application information for one more set of weights. A happy medium will arrive fairly quickly, as gains of accuracy from the model diminish and demands for application parameters rise. An interesting study by Wang *et.al.* statistically demonstrates that among the 24 LFK (Livermore loops) benchmarks, there are but two to five predictive

dimensions: A few scores should characterize a machine [WAN88].

2.3.1 Other Architectures, Other CUTs. In addition to the three modalities used in Figure 1, the earlier table of modalities has a fourth, which contrasts processor-to-processor messages against intra-processor memory references. Because these modes are so disparate in speeds, they can serve as the first differentiation in a CUT for an array processor. Communication may proceed up to three decimal orders of magnitude faster direct to memory. The observed effect can often be a factor of ten in program execution [RED88].

2.3.2 One Simplification. There is a compression of the CUT graph that some circumstances encourage: The whole graph is reduced to a table of equal-gain performance increments. The following must hold for the method to work:

1. The architectural layout is *fixed*, i.e. the underlying tree remains the same.
2. The application is also *fixed*, so that that frequency weights on the tree cannot change (these characterize fully any application).
3. Computational capacities (admittances) **can be changed within limits**. This amounts to varying the implementation via faster components, better subunits, or less expensive pieces. The limit on improvements is that a mode cannot "amplify" capacity, i.e. exceed an admittance of 1.

For comparison, choose a base version of the architecture XXX. Its times are computed from the leaf entries; they are then adjusted so that the computed coefficient is 1. Simply multiply each contributed time by the actual efficiency coefficient. From leaves in Figure 1,

"t(A)" = (.15/.07)*0.194 = 0.415
"t(B)" = (.15/.1)*0.194 = 0.291
"t(C)" = (.35/0.3)*0.194 = 0.227
"t(D)" = (.35/1)*0.194 = 0.068

These "normalized times" are working values for computing a simplified scheme. In the following example, the "times" have been multiplied by one hundred and rounded or truncated, with the understanding that the sum divides into a hundred, rather than unity. Integer values (41, 29, 23, 7) are convenient to work with.

Because factors B and D are at their best performance, only A and C are examined further. If degraded performance were of interest for B and D, they too would be in the final table. Note from Figure 1 that the best (minimal) time A can have is 29, the same as B. Factor C similarly can improve until it equals D at 7. (Assume that the CUT is built with factors of interest as leaves.) Both A and C can be as slow as one might imagine, although Figure 2 restricts its descriptions of these worst performance cases to practical ranges.

	Factors			
	A	B	C	D
Base Time (Normalized)	41	29	23	7
Minimal Time Possible	29	29	7	7

System XXX: Allowed Normalized Times

Since A can improve by -(41-29)= -12= -(3*4) "time units", and C by -(23-7)= -16= -(4*4) units, table entries can be implicitly in steps of 4. That is, "normalized time" = (A+B)*4+100, where A=-3, -2, -1... and C=-4, -3, ... Circumstances will dictate the exact linear reformulations appropriate to other CUTs.

2.3.3 Use of the Table. The expression in Figure 2 provides an index of computation speed for a new machine variant *relative to the base implementation* of the understood, fixed architecture XXX running *the chosen application*. This simplification is especially useful whenever an application is prominent, preferably dominant, in an environment. Good money estimates for subcomponent substitutions further improve the method's utility. Here is an example. Suppose there is a machine like XXX, but with a huge amount of real memory (A=-3). Unfortunately, its loader produces clustered references (C=+2). The performance of this "XXX?" machine relative to XXX *and the application* is 100/[(-3+2)*4+100]= 1.04, which hardly seems to justify its added memory cost. A improved loader would probably make it a more competitive product.

The tableaux work not only for digital computer modeling, but serve equally well for other engineering practice, such as simplified aerodynamic drag coefficient estimation [WHI69].

2.3.4 Refining a CUT Component. In many cases the chosen dimensions of an architecture admit numerous parameterizations. A gross comparison, such that of s against v, may simply use peak benchmark performances to establish each mode's fundamental rate. Nonetheless, it can be revealing to perform a parameter variation within a mode while other factors are constant. A typical example of this is the test VECOPS from Los Alamos National Laboratory [BUC84, Table I ff.]. VECOPS performs a variety of binary and triadic vector operations while varying vector length considerably, from 10 to 10000 words. Vector performance may vary by one and a half decimal orders of magnitude over the domain. Programmers know that good performance on shorter vectors renders a machine more flexible in running various algorithms. Each significant vector length can be an arc label in a part of a CUT representing vector capabilities.

Communication and memory are also open to refinements in their characterization [GRU86, LYO88, RED88]. The test set in the latter half of this text addresses communication; it has uncovered special details that might otherwise cause difficulty in program performance. The set seeks weaknesses in common application layouts (as opposed to predicting exact performances). It is preventive and diagnostic in nature. VECOPS can be viewed in a similar light. Both warn programmers of which length vectors or messages perform unsatisfactorily.

2.4 Architectural Focus Overall

There are several recommendations for architecture-focused estimation, in order of increasing difficulty:

1. **Establish** values for the six general capacities sketched by Hillis [HIL85].

2. **Identify** logical (virtual, language level) operations of interest. Note that this step, although innocuous, is not entirely independent of applications. For instance, including Boolean operations sanctions some classes of application, and excluding floating point certainly precludes others.
3. **Extract** the competitive modes of the operations in 2, leaving the categories in reduced state (single mode of execution with predictable time, given operands). Finding some modes is much eased if compilation can be performed with user-controlled degrees of optimization (including none). This promotes parametric benchmark scans that might otherwise be difficult.
4. **Sketch** a CUT model in outline.
5. **Run** basis benchmarks and characterize modalities.
6. **Generate** optimistic estimators via benchmark results and Amdahl's law. These estimators set an upper bound on the performance envelope, since they assume a loss-less mixing of the competitors.
7. **Refine** basis measurements of dominant modalities to emphasize significant variations. This provides an idea of how restricted the views of performance are.
8. **Complete** the CUT model. If appropriate, simplify the model to tabular form.

The role of applications is much diminished in this architecturally-focused approach, but it is not absent, certainly not in the CUT. The same is true of benchmarks used in the process. Besides representing an implementation parameter (capacity), each basis benchmark implicitly qualifies overall performance estimates from the way the benchmark is written. The degree of restriction depends upon the style, focus and need within the estimation task. The qualifying reflects implicit or deliberate influence from the applications domain. Fortunately, the comparisons have only to deal with a few application extracts.

2.4.1 Summary and Problems. CUTs provide a resilient framework that displays elements of computer performance estimations. Principal CUT strengths are explicitness and malleability. Each part of the formulation has a significance that can be interpreted and, within limits, manipulated. The tree structure of the CUT reflects important system architecture. Frequency weights f_i serve to define application classes, and capacity weights c_i describe a system's implementation. All these details are explicit and subject to comparison and change. Variations on the frequency weights generalize what is found in the literature on benchmarking interpolation [BUC84]. Changes in capacity admittances provide interesting views of implementation gains (Figure 2). It appears that the most difficult of the contrasts will be with architecture--this involves transforming one tree into another, or in some other way establishing CUT comparabilities. While a CUT model reduces comparisons to graph-equivalence questions, this in no way alleviates difficulties of the task. A second open question is the actual constructing of a CUT. Automatic rules of construction would shed further light on the evaluation of systems.

3. Speedup and Scaleup in Parallel Processing

Benchmarks eventually have to be interpreted. It is important in parallel processing that unambiguous terminologies be used, as in [GOT84], to keep interpretations meaningful. The term *speedup* seems intuitive and well established in multiprocessor computing [88SIA]. In essence, a problem's original execution time is divided by its new time on a more powerful configuration. It is hard to quarrel with this simple formulation; use of the term is widespread. However, there is a shift in terminology that is occurring regarding *speedup* versus another parallel processing technique that will be designated *scaleup*. For example, the recent claimants of the Bell *Speedup* Award [88SIA] actually espouse what is here defined as *scaleup* [GUS88].

Let $N_{n,p}$ where $0 \le N_{n,p} \le 1$ be some suitable coefficient of effectiveness that, for problem size n and processors p, expresses the efficiency relative to a user's need. A typical formulation for $N_{n,p}$ is overall program speedup divided by the number of processors, p. This formulation expresses the recovery of hardware investment. The major interest is that $N_{n,p}$ varies with two parameters. Parameters n and p can vary either together or separately. The following limits serve to emphasize the definitional points (in the extreme):

Loadup limit ($N_{n\to\infty}$). *Loadup* depicts a fixed-capacity machine that must run ever-larger variants of the same problem. A suitable problem decomposition, one where computation eventually dominates communication, will give an excellent indication. The idea of loadup limit is of less immediate interest, and is included mostly for completeness.

Speedup limit ($N_{p\to\infty}$). *Speedup* is the common, conventional term for parallel processing improvement. The problem size n is fixed and the number of processors is allowed to grow.

Scaleup limit ($N_{n,p\to\infty,\ n/p=k}$). *Scaleup*, which is actually linear scaleup when n/p=k, is often what many users want without clearly stating such [GUS88]. Both problem and machine grow proportionally. A typical example is when current versions of some mesh-connected problem run well enough, but solutions for larger problems are desired. A larger machine is purchased to perform this new assignment in the same time as previously.

3.1 Secondary Use of Definitions

Exaggerations via speedup and scaleup limits may widen intuitive perspectives. This is demonstrated on two advanced architecture designs. Dataflow in many aspects addresses speedup, and massive parallelism, scaleup. The correspondence is also reflected in everyday experience with less venturesome parallel processors.

N. Wirth has proposed a epigrammatic description of programs that can be paraphrased as:

serial program --> *control-graph* + *data-objects*

This widely accepted formulation can be transformed using the concepts of speedup or scaleup. Take, for

instance, a dataflow machine:

(i) **dataflow program --> speedup{decompose(*control-graph*)**
+ value-dependencies(*data-objects*)}

Similarly for massively parallel machines and scaleup:

(ii) **massively-parallel program -->**
scaleup{add_obj_bitscans(*control-graph*)
+ bitdefs(*data-objects*)}

Formulation (i) asserts that dataflow divides a program into ever-finer computations. There is certainly a ring of truth in this, since a principal emphasis is to provide many threads of computation to offset sundry latencies of scale. Because resources are constantly increasing, computational granularity must shrink. Even with the mildest overheads, the fine grain can lead to trouble, e.g., low-level bottlenecks may be rather widespread. Eventually, the reduction of granularity causes single operations to be scheduled--the mechanics of doing this well can be expensive for general purpose machines. Eventually some minor overhead becomes major, grain being sufficiently reduced. Data-flow token matching is one example which has recently seen very interesting proposals [IRA88].

Scaleup admits replication of detail with little loss of overall effort. One way to interpret the massively-parallel paradigm is to imagine a new bit-level dimension along what were formerly program atomic elements (e.g., long floating point becomes a vector of 64 bits.). Scaleup then admits broadcasting of bit-iterations to processors which compute along the new bit dimension. However, lockstep processors are not the only architectures for which scaleup appears promising. One following example, the ring with chordal shortcuts, scales up perfectly on a coarse grained hypercube with MIMD organization. What does not work for this particular hypercube is speedup. Message-passing latencies (overheads) are simply too great.

4. A Test Set

An earlier study to which this work is sequel stressed embedding performance benchmarks into comprehensive frameworks that set their context of use and validity of interpretation [LYO88]. The example chosen in [LYO88] is that of process communication, which is absolutely necessary for parallel processing (at the process level). The examples that follow extend the beginnings seen earlier, following the framework that was proposed. That framework has two dimensions. One delineates communication dependencies in a somewhat rough-hewn manner as processes that can be scheduled (i) nearly independently, as in radiation transport, (ii) locally-dependently, *a la* fluids, and (iii) globally interdependent, like molecular dynamics problems. A second dimension measures degrees of abstraction away from the physical hardware itself: hardware, instruction level, processes, algorithms and applications. As indicated, processes have been selected for study. Processes are a common level of parallel computation for contemporary MIMD machines.

The immediate role of the communication benchmark set is to inform a programmer trying to use a new machine. As will be seen, the benchmark programs cause some machines to display unexpected behavior; it is better that a programmer know about this prior to designing his algorithms for a project, rather than learning from trial and error.

The set provides several sets of parameters that can be explored in the spirit of mode refinements mentioned earlier. Some of these, such as polling frequency, can be quite sensitive on certain operating systems. Again, the programmer should know this. Other parameters, such as message length, can cause certain fixed-frame length systems to crash if messages are too short and too frequent.

Numerous versions of the programs exist to accommodate distinct architectures. In one test, the ring structure, the degree of global dependency can be relaxed; this has been useful in investigating the scaleup capabilities of hypercube systems.

4.1 High Interdependencies: A Ring Model

One can define a logical (but hardly unique) communications structure at the process mechanism level. The role of each logical structure is to provide an abstract model of communication divorced from fine details of (i) the original problem, or (ii) algorithmic and coding features not related to process communication. (Communication denotes both synchronization and data transmission.) Certainly there may be numerous acceptable models for a given application. However, to study process communications, specific examples must be chosen and tested; a ring structure has been selected to exemplify global process dependencies. It is implemented with parameter variations as follows:

A. Synchronization (busy-wait; polling; interrupts)
B. Mode of transmission (by-value; by-reference)
C. Message length (short to long)

Further variation is necessary within gross parameter selections. For example, polling introduces the notion of *frequency* which must be explored. Computation per datum should be adjustable as a parameter. In addition, a variance about this computation can be set by another parameter.

The synthetic ring benchmark for global dependencies works as follows: Each of n logical nodes will originate x messages, and additionally, process all other messages passing by. The number x of messages and their length y are parameters. Each message travels around the ring while being "processed" synthetically by each node. As a message returns to its origination node, it is removed from the ring traffic. A new message is sent unless all x have been sent. When all nodes have sent and received all of their messages, the ring of processes is dissolved and the results are reported. Communication is asynchronous, with message traffic regulated by a simple form of flow control. This keeps slower nodes from being overrun with messages. Such control is essential on systems that cannot control buffer overflows. Messages are acknowledged on a one-to-one basis. Thus, at most, a process (node) will have one waiting message.

A variant of the ring with chordal bypasses has been used to investigate *scaleup* on MIMD message passing system, e.g., hypercubes. The chords allow a message to pass through k nodes then return to its origin, rather than having to complete a whole circuit. Thus a node directly influences only k nodes (although k is a parameter). There are overlapping spheres-of-influence, a refinement of completely global interactions. Results will show that the modified ring scales up excellently.

4.2 Low Dependencies: Random Communication Model

Another synthetic benchmark for process communication depicts computational objects (nodes) that can be scheduled nearly independently. This circumstance might be radiation transport, or within a computer system, storage scavenging on linked lists. It represents the other end of the interdependency spectrum from the ring. The algorithm implements a pool of n logical nodes (processes), from each of which random messages are initiated, dispatched, received and processed. The communication patterns for this algorithm are depicted in Figure 3a. Each node in the graph has n-1 (or n if local communication is set) edges, each of equal weight in the logical structure.

The benchmark consists of two principal sections, a manager (or host) node and a logical collection of worker nodes. Nodes process some workload and randomly distribute other chunks of their workload to other nodes. Each node is initialized with an equal amount of data, hence the workload is initially balanced.

A node begins and continues processing its data until it is notified within its workset (via a special value) to *pass* some of it on or until it finishes that set of data. When it must yield part of its present workset, it randomly chooses (based upon data values) a destination node and determines how much to dispatch. Whenever the node completes its assigned data, it checks its message queue for more work. If there is a pending message the node records the work completed and services the next message. Otherwise the node reports to the manager. An idle node awaits messages. When the manager has recognized that the entire workload has been processed, results are recorded and the pool of nodes is dissolved.

As earlier described with the ring benchmark, this algorithm on shared memory machines allows for various synchronization methods; they again are busy-waiting, polling and interrupts. The protocols used on message passing systems are the send/no-wait and the blocking-receive routines. In addition, computation granularity per datum is adjustable, as is the frequency of message originations (work dispatching).

4.3 Local Dependencies: A Mesh Model

A middle ground in the interdependency spectrum has models in which the next state is a function of a small number of neighboring processes. The model chosen to represent this has a two-dimensional mesh structure with nearest neighbor communication dependencies (Figure 3b). A description of the algorithm follows.

An N by M mesh with toroidal (wrap-around) connections is created with N*M logical nodes. Each node in the mesh communicates with four nearest neighbors. A mesh point $\mathbf{n}[i,j]$'s nearest neighbors in a N by M mesh are defined as $\mathbf{n}[i,(j+1) \bmod M]$, $\mathbf{n}[i,((j-1)+M) \bmod M]$, $\mathbf{n}[(i+1) \bmod N,j]$ and $\mathbf{n}[((i-1) + N) \bmod N,j]$. The toroid eliminates any scaling effects of boundaries.

The program iterates state-by-state, and each state has two distinct phases, communication and computational. The communication phase does information exchanges and node synchronization. The computation phase defines the amount of computation performed by each node (process) between synchronizations; the granularity of this phase is an important factor in design and software costs. Generally, coarse grain synchronization is desired, but often is difficult to implement. It may not be available in some applications. The mesh program stops after performing a given number of state iterations.

Communication places a global constraint on the overall computations. A node (process) needs values obtained from the previous states of its nearest neighbors to continue; this fact prevents it from outdistancing its neighbors. However, the cohesiveness of the process states is different for various programming approaches; these deviations will be described shortly. Since processes are periodically required to wait for other processes, the mesh benchmark can be classified as an iterative synchronous algorithm.

Parameters for the mesh include message length and computation granularity per datum. On shared memory machines three types of synchronization methods are available; busy-waiting, polling and interrupts. The notions of synchronous versus asynchronous (or chaotic) algorithms must be distinguished from synchronous and asynchronous communication; hereafter the terms will refer to communication (unbuffered=synchronous, buffered=asynchronous). Another issue is that of mapping the mesh onto the hypercube topology while avoiding arc dilations.

4.3.1 Synchronized versus Chaotic Approach. An iterative algorithm's interaction points can often be implemented as either synchronized or chaotic on message-based systems. The synchronous approach on message passing multiprocessors ensures that processes are all proceeding on nearly the same round (time step) at any given point. This is implemented by sending acknowledgement messages after the receipt of a data message. A process can not begin the next communication phase until it confirms that its neighbors received its previous message. Since the time taken by a stage of a process is a random variable, the synchronized approach has the drawback that some processes may be blocked at a given time; performance of the algorithm is degraded. Synchronization overhead, seen in acknowledgment messages, is another encumbrance of this approach.

One way to reduce these bottlenecks relies upon a less-structured computation. One process may start the next state before others have finished. The cost of such a scheme is often an increase in the complexity of the synchronization structure. This in turn produces higher implementation costs and memory requirements. In the chaotic algorithm, processes continue to execute as long as some new information is available. Since it is possible in such a scheme for one neighboring node to issue two data messages before another neighboring node issues its first to a given node, a monitoring system has to be developed to ensure a consistent sequencing of events.

4.3.2 Mapping a Mesh onto a Hypercube Topology. The nearest neighbor mesh has a well-defined structure with a North-South-East-West pattern of communication. Mapping the mesh structure perfectly onto the hypercube may result in substantial saving in communication time. As explained by Saad and Schultz [SAA88] (for a similar structure), an excellent example is that of mesh geometries that arise from the discretization of elliptic partial differential equations in one, two, or three dimensions. Most iterative

methods for solving elliptic PDE's require only local communication, i.e., communication between mesh points that are neighbors. If the mesh is perfectly mapped into the cube, then only local communication links will be required between nodes of the hypercube. This produces important savings. This economy, however, is a function of communication capabilities and program function. Its effects may be dominant or insignificant. The synthetic mesh benchmark tests for this.

Two approaches for mapping are distinguished, *direct* and *non-direct*. The number of physical links that a logical path has to traverse defines the *dilation* of that logical path [NI87]. In direct mapping, each logical path incurs one dilation; this is optimal when achievable. However, a direct mapping can be difficult to establish; for some problem topologies it is NP-complete. In the non-direct method, processes are mapped onto the hypercube in the most convenient way (usually reflecting the logical structure of the algorithm), ignorant of the underlying architecture. This, however, leads to greater average path dilation, which increases the number of logical paths that traverse a particular physical link, referred to as the *sharing* of that link [NI87]. Path dilation and link sharing directly reflect communication delay. Direct mapping minimizes these components, whereas the non-direct approach does not. An experiment comparing direct mapping and non-direct mapping is investigated for one particular hypercube. Results appear later. (See the "Summary of Test Results" table, near the end.)

The direct mapping used here implements an extension of a scheme developed by Saad and Schultz [SAA88]. The mesh model demands wrap-around communication paths for two-dimensional grids for even-valued N and M, each greater than or equal to four. Note that these requirements constitute a hypercube of dimension ceiling(log2[N]) + ceiling(log2[M]).

4.4 Benchmark Synchronization Mechanisms

The notions of synchronization and communication are difficult to separate because communication primitives can be used to implement synchronization protocols, and vice versa. In general, communication refers to exchange of data between different processes. Synchronization is a special form of communication, in which the data are control information [DUB88].

4.4.1 Synchronization via Shared Variables. Two types of synchronization are commonly employed with shared variables. These are *mutual exclusion* and *condition synchronization*. Mutual exclusion ensures exclusive execution of critical sections via control of process scheduling. Condition synchronization occurs within a set of cooperating processes when a shared data object is in a state that is inappropriate for executing a given operation. Any process that attempts such an operation should be delayed until the state of the data object changes to the desired value as a result of other processes being executed [HWA84]. The strategies (i.e., synchronization methods) used for this delay period on shared memory machines are busy-waiting, polling, and interrupts.

Busy-waiting is a form of synchronization which uses processor cycles to test a variable until a desired value occurs. This is implemented using a loop structure (e.g., while(*condition* = **false**);), which spins in a tight loop until the proper condition arises. Polling is similar to busy-waiting in that a process has a loop checking condition; however, when it does a check and the condition is not the desired value, the process sleeps a fixed interval before checking again. This is implemented using the Unix *select*

statement (e.g., while(*condition* = **false**) select(*time_interval*);). Polling has the advantage of not wasting the processor during the specified sleep interval. A disadvantage arises when the desired value is set during process sleep. Therefore, the specified time interval is integral to a polling implementation. Interrupts are implemented with the Unix *signals* capabilities. With this method, a process sleeps until it receives a signal (interrupt) to awaken. As with polling, interrupts require the services of the operating system. Busy-waiting does not.

4.4.2 Synchronization via Message Passing. On loosely coupled systems process interaction is accomplished through message passing. Communication is achieved through message passing since a process receiving a message is receiving data from another process. Message passing is also a form of synchronization, since a message can be received only after it has been sent [QUI87]. The protocols used by the communication algorithms are the send/no-wait and blocking-receive routines. The send/no-wait (i.e., non-blocking) statement never delays the further execution of the invoking process; messages are buffered. The blocking-receive will suspend execution of a process until a message is read. For these algorithms this mechanism does not hinder execution, since no useful tasks can be performed during this phase.

5. Results

This section relates experiences and results for the communication benchmark set. The examples illustrate typical uses, but are not comprehensive. The results illustrate vast performance differentials obtainable with various (1) architectures, (2) synchronization mechanisms, (3) message and computation granularities, (4) synchronization granularities, (6) mappings to physical structure, and others. This underscores the importance of establishing the characteristics and capabilities of a machine's communication subsystem.

Experiments cover both message passing and shared memory multiprocessors. The message-based machine is a five dimensional Intel hypercube. The tightly-coupled systems include a twenty processor Encore Multimax® and six, sixteen, and twenty-four processor Sequent Balances®, plus an Alliant FX/4 used by colleagues at the University of Oklahoma [LAK87, LAK88].

In the following examples the notion of computational granularity must be distinguished. In the ring and mesh models grain size refers to the amount of computation performed between communication interactions, which are known. In the random model it determines the computation performed for each data item a node encounters; this is because communication is triggered at random points in the data stream. The compute/communication *balance* is stochastic; consequently, grain is best defined on a per-datum basis. Therefore, the meaning of the terms *fine* and *coarse* grain are algorithmically dependent notions, with the random model somewhat different for convenience.

5.1 Sample Results from Ring

An earlier report [LYO88] gave a first view of the benchmarks via experiences with the ring benchmark alone. A number of graphs in the earlier work show variations for changes in synchronization methods, by-reference or by-value process communication, and scaling. Results for polling were especially unpredictable on several systems, and for this reason, have been more thoroughly explored, below.

5.1.1 Hypercube Scaleup. The effects of scaling, and especially *scaleup* as defined earlier, were pursued with a ring modified with chordal bypasses. This ring was placed on hypercubes of growing sizes as the ring grew at the same rate. Since the chords provide localities of computation that are independent of overall ring size, the chordal-ring program runs essentially at the same time-to-completion independent of the ring size. This is true, naturally, because as the ring is made larger, the number of processors grows proportionally. While quite distinct from speedup, scaleup is apparently [GUS88] rather common in engineering and scientific computation. Figure 4 depicts a typical ring scaleup experiment. Clearly, for suitable problems, the hypercube scales very well. There is one caveat, however. A real problem may have convergence conditions which propagate slowly from neighborhood to neighborhood, thus delaying program termination. The chordal ring model does not duplicate such conditions.

5.1.2 Polling and Shared Memory Machines. Numerous experiments were conducted studying the effects of different polling levels on selected shared memory machines. Performance levels vary significantly from one architecture to another as well as within the confinements of an architecture. Figure 5 illustrates this point: Here time-to-complete is plotted against polling frequency in milliseconds for various ring sizes.

When the ring size is significantly larger then the number of processors (such as a 30 node ring on a six processor Balance 8000, Figure 5a), performance levels vary considerably. The polling frequency produces expected results when the ring size equals the number of processors (e.g., a 6 node ring on a six processor Balance 8000, Figure 5b). Note for this example and others, parameters were adjusted to obtain comparable timings. The fluctuations become apparent for an 8 node ring (Figure 5c) for this particular shared memory machine. Similar observations are obtained on a twenty-four processor Balance 21000 for a ring of size 30 (Figure 5d). Abrupt transitions are still evident in 5d, but not to the same degree as experienced on the smaller machine (5a). The 5d ratio between processes and processors is much smaller. On a twenty processor Multimax with a ring size of 30, no observable fluctuations are apparent. However they were apparent on a ten processor Encore Multimax in a study performed by Lakshmivarahan and others using the same code [LAK87]. For a ring size of 10 there were no significant fluctuations, a slight deviation as the ring grew to 20, and prominent variance for 30 nodes. A later study on an Alliant FX/4 revealed a similar phenomenon [LAK88].

The pattern of fluctuations differs among the various architectures; however, there is an inherent trend over the same architecture and across the test programs, at least for one shared memory machine. Consider the results obtained on the Sequent machines for both the ring and random communication algorithms. For random communication, transitions (on the Balance 8000) were most evident at polling frequency intervals of 19 ms to 20 ms, 29 ms to 30 ms and every 10 ms step tested thereafter. This pattern is duplicated for the ring (30 nodes). In addition, the pattern remains when both programs are ported to a larger system of the same type. It is apparent on these shared memory machines that process/processor mismatch amplifies performance differences that result from choices of polling frequency. The cause of this anomaly seems to originate in the scheduling algorithm for these systems; when the number of processes for a problem exceeds processors, scheduling is erratic.

5.1.3 Hardware Measurements of Unix's Select Statement. As shown in Figure 5, polling offers a wide range in performance levels with little change in polling frequency. Polling as stated earlier is implemented via the Unix *select* system call. This next experiment focuses on services times recorded for this statement in conjunction with a typical benchmark run. Service times are captured with the use of our NIST measurement system, which obtains low level measurements with timestamp perturbations of only 3-5 microseconds [CAR88]. The following experiments were performed on a six processor Sequent Balance 8000.

In normal operating mode, the average service time for the select statement follows that of the system clock *tick* (10 milliseconds). As shown in Figure 6b, service times for a small ring follow this expected path and as a result performance levels are predictable (Figure 6a); execution time increases with polling frequency steps. For this problem it is easy to identify an optimal polling frequency. Small polling intervals work best. However with the 30 node ring (Figure 7), polling service times vary significantly, as do performance levels. This is especially true for the interval of 19 ms to 20 ms, where service times dropped moderately, triggering a drop in execution time. This forementioned pattern is not evident in subsequent polling frequencies, where execution time fluctuations do occur. In this implementation, polling service times are much greater than the requested poll, especially in the interval up to 19 ms. In some cases it is over twice as much as recorded for the smaller ring. For example, (Figure 7b) when a poll of 10 ms is requested the actual average poll received is 23 ms. After the readjustment at 20 ms, service times are only slightly higher than the requested poll. In contrast to the smaller ring, completion time for the larger ring decreases with higher polling frequencies.

From the previous observations it can be concluded that for this shared memory machine, time-to-complete is a function of service times for the select statement in polling implementations. Irregular service time deviation is only noticeable when the model contains a process/processor mismatch. As stated earlier the cause may be linked to a feature in the process scheduling algorithm. Figures 6 and 7 show typical extremes moving from one process per processor (Figure 6) to many (Figure 7). Intermediate ratios show intermediate distortion. The absolute size of the ring is an additional influence; larger rings highlight any scheduling delays.

5.1.4 Locked vs. Dynamically Assigned Processes. In an environment such as Sequent Dynix, a process runs without interruption on a processor until it blocks, terminates or is preempted by another process with greater or equal priority [SEQ87]. In the ring with 30 nodes, processes are often pre-empted and rescheduled on other processors. From experience, this seems to affect the polling implementation adversely, which consequently reduces performance. The next experiment aims at reducing pre-empted processes, thus alleviating any erratic scheduling. In this experiment, sets of unrelated processes in the ring are bound to specific processors (via the tmp_affinity system call). A bound process runs on no other processor. For a ring size of 30, five processors are assigned six nodes each, leaving one processor free to do other system chores. As shown in figure 7a, this implementation exhibits increased performance and predictability. Some cases run over five times as fast for exactly the same program code and parameter set (cf. Figure 7a). Another point of interest is the increase in polling service times (Figure 8b). Services times range from 103 ms to 130 ms, much greater than recorded for unbound processes. In addition, there is a very high variance in service times. However, in this controlled environment, service times and their variance have little appreciable effect on execution times, contrary to earlier discussed implementations.

5.1.5 Conclusions for Ring and Polling. It is clear from the previous observations that a programmer must be aware of subtle characteristics in system software, especially scheduling and processor assignment. Seemingly small polling parameter variation yields profoundly different performance levels. It seems that variance in results as a function of polling frequency is inherent in shared memory architectures. This variance is amplified as a process/processor mismatch grows. Yet for one particular system, a significant increase in polled performance is achieved by binding processes to processors; locked processes do much better than dynamically assigned processes.

5.2 Sample Results from Random Communication Model

5.2.1 Hypercube. On a hypercube where each node has a dedicated processor, linear speedup is only obtainable for the random model when communication between nodes is non-existent. The initial load is perfectly balanced and remains undisturbed. As expected, adding any degree of communication (defined here as data transmission) and thereby altering the data allotment (load balance) degrades performance. This degradation can be substantial, as illustrated in Figure 9. Speedup is almost halved at a very low communication frequency, at about .01% of items causing messages. In this case, data is shifted away from some processors onto others. Because of infrequent communication, the model never regains suitable load balance. Hence, some processors starve while others are burdened with heavy workloads. Execution time is established by the slowest of the nodes. Performance improves as the communication rate increases up to two percent. This can be attributed to better load balancing, a benefit that dominates any penalty from increased communication traffic.

As communication becomes too intense (3.5% for this data) the system fails, as indicated at several points in the graph. This failure is primarily caused by the system imposed limit of message buffers, coupled with the communication structure of the algorithm. In this model, the send/no-wait primitive is used for communication. This allows the sending processes to get arbitrarily far ahead of a receiving process. This leads to nodes accumulating messages at a rate well beyond the capacity to process them. Whenever new messages exceed a node's buffer capacity, they are (unfortunately) lost. Deadlock results. It appears impractical to program against this, indicating that related--communication intense--applications may be difficult to support.

5.2.2 Shared Memory. On tightly coupled multiprocessors, *messages* are buffered in a large shared memory; they are not vulnerable to buffer depletion as on the hypercube. As the frequency of messages increases, communication cost should come to dominate and performance to decline. However this is not always reached. When computational granularity is coarse, additional communication does not dominate and performance remains relatively constant (Figure 10; for *coarse grain(CG)*).

Experiments were conducted for the various synchronization mechanisms for both *fine* and *coarse* grain data sets (since polling results follow that of busy-waiting, they are only shown when they differ significantly). These tests are performed on a sixteen processor Sequent Balance 21000. Node collections of 12 and 24 are chosen to represent a system both without contention (12<16) and with (24>16).

When few computations are performed for each data item, communication overwhelms the memory network; performance is disastrous, especially for interrupts (Figure 10; *fine grain(FG)*). For very low communication frequency, interrupts perform quite well, but as the frequency increases performance rapidly deteriorates. As expected in the model with 12 nodes (no processor contention), busy-waiting is best.

As mentioned, the performance difference for coarse grain problems is minimal over the spectrum of communication frequencies. However, as with hypercube performance, infrequent communication often leads to load imbalance that degrades performance. The best of shared memory performance arises from a moderate communication frequency. Interrupts outperform busy-waiting for coarse grain data sets in a system with processor contention. This is contrary to other implementations in this example. Interrupts free a processor when no useful work can be performed, whereas busy-wait consumes a whole timeslice. Furthermore, in this domain of infrequent communication, efficient use of the system is not compromised by communication overhead. A coarse computation grain masks system overhead. As a result, the 24 node model outperforms the 12 node model. As a last observation, 50-millisecond polling for this data yields maximum processor utilization and best overall performance.

5.2.3 Conclusions for Random Communication Model. On the hypercube, poor performance is exhibited for very low communication frequencies; this can be attributed to unbalanced workloads. Because of the limited buffer space, communication intense applications of this sort may be difficult to support on such systems. On shared memory machines, the cost of communication for fine grain data sets is increasingly expensive for random communication. Conversely, data balancing is the dominant factor in coarse grain problems; moderate communication frequencies are required to obtain this balance.

5.3 Mesh Results on Hypercube

5.3.1 Short vs. Long Messages. This hypercube experiment explores effects of different message lengths, the number of time-step iterations, and strategies for synchronization and mapping. The amount of information passing through the communication subsystem is a constant. This constant is formulated by the product of message length and iterations; thus shorter messages require more synchronization stages. The lengths of the messages range from short (256 bytes) to the system's limit (16K bytes). The results shown in Figure 11 are for a 4 x 4 mesh.

It is clear that longer messages work better over the spectrum of implementations. In addition, performance levels deviate little for message lengths of 1K bytes and longer. However performance beneath this 1K byte threshold drops off significantly. This is most evident for the synchronous non-direct mapped version, whose execution time almost doubles for the interval of 1K bytes down to 256 bytes. This performance degradation can be attributed to the system sending information along its communication network in full-length packets of 1K bytes. Messages less than 1K bytes still ride in packets of 1K bytes. Given this, it is naturally better to send fewer but longer messages. However, this may not be practical or even be possible for an application. Therefore, it is important to reduce the frequency of communication in problems requiring short messages.

5.3.2 Synchronous vs. Asynchronous. Frequent short messages stress the communication capabilities of an architecture, as demonstrated in Figure 11. In this domain, the chaotic calculation easily outperforms synchronous versions. In this example the chaotic approach runs approximately 22% faster than its counterpart. In this test the computational grain is uniform for each process at each stage. However, when the computation per stage is a random variable, the performance gap is narrowed. The asynchronous version runs only about 11% faster (not shown in graph). For an implementation with very fine grain synchronization and message length, performance differences of over 50% are observed.

For long messages, chaotic and synchronous performances are similar; times are within one percent or less. Longer messages effectively elongate the period between synchronizations, thus reducing the significance of process-blocking in the synchronous versions. Hence, performances are comparable.

5.3.3 Mapping Strategies. As expected, for the spectrum of message lengths tested, direct mapping is more effective than non-direct (average dilation in this example is 1.5, see Figure 11) for this particular architecture. For medium to long messages, various mapping strategies have little effect; the performance difference is only about three percent. This is due in part to the longer computational phase between synchronizations, which reduces the percentage of time spent communicating. The performance difference for short messages is slightly higher, approximately eight percent. On this system, mapping a program's communication structure to optimal communication links on the architecture yields only modest gains, and in many cases may easily be ignored.

One major vendor has introduced message routing via hardware communication circuits, thus reducing message transmission time substantially. With uniform latency and bandwidth throughout the system, the underlying topology becomes transparent, thus supporting a wider range of applications and easing the task of programming. But performance will always differ for various architectures and algorithms. The programmer must carefully examine his own requirements for a particular problem on a given machine.

5.3.4 Conclusions for Mesh. Long messages work best over a broad scope of applications. However long messages may be difficult or impossible to realize. Thus, performance for short messages is important. The asynchronous approach for short messages can run up to 50% faster than synchronous algorithms. However, when the workload for each process at a stage is not uniform, this performance gap is halved. Only modest gains are achieved by mapping the communication structure to the underlying topology.

5.4 Summary of Test Results

The table below is a amalgam of results previously discussed. It highlights potential problem areas, as well as good architectural features for programming. Note that the results represent a sample of tests available from the communication benchmark set, and are not comprehensive.

Summary of Test Results		
Model	**Coupling**	**Remarks**
Ring	Shared Memory	-variable results from polling frequencies is inherent? -scheduling erratic when #(processes) > #(processors) -locked processes superior to those dynamically assigned
	Hypercube	-scales up well if each logical node has physical node -logical structure amplifies any random variances
Mesh	Shared Memory	**results not available at this time*
	Hypercube	-long messages work best over a broad scope of applications -the asynchronous approach for interaction points is especially efficient for short messages -perfectly mapping communications to underlying topology may yield but modest gains
Random	Shared Memory	-communication for fine grain problems is increasingly expensive -load balancing is main issue for coarse grain problems
	Hypercube	-poor performance exhibited at very low communication frequencies; attributed to unbalanced workloads -communication-intense applications can be difficult to support

5.4.1 Acknowledgments. Vivian Lawrence read an earlier version of the text and suggested numerous details for improvement. Carl Smith reviewed a separate note, part of which became the CUT graph section. Argonne National Laboratory, Argonne, Ill., and the Supercomputing Research Center, Lanham, Md., kindly provided systems for measurements.

6. References

[87SEQ] **Guide to Parallel Programming**. Sequent Computer Systems, Inc., Beaverton, Oregon 1987.

[88NIST] *Private conversations with D. Bailey and J. Gary*. The range combines two informal estimates, one of 3x and the other a range of 3.5x to 6x. These numbers should not be taken too seriously, but burdening by SCATTER-GATHER practices certainly should. On the other hand, with sparse matrices, it is better to have SCATTER-GATHER than not. See the paper by van Waveren [VAN87].

[88SIA] Gordon Bell Awards for Parallel Speed-up. *SIAM Activity Group on Supercomputing* **Newsletter** (First Quarter, 1988), 1-2.

[AMD67] Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. *Proc., AFIPS Spring Joint Computer Conference 1967*, Atlantic City, N.J., April, 1967, 483-485.

[BAI88] Bailey, D., Brooks, E., Dongarra, J., Hayes, A., Heath, M. and Lyon, G. Benchmarks to supplant export "FPDR" calculations. NBSIR 88-3795, June 1988, 20pp.

[BUC84] Bucher, I.Y. The computational speed of supercomputers. Report LA-UR-84-740, Los Alamos National Laboratory, Los Alamos, New Mexico, 1984, 15pp.

[CAR88] Carpenter, R.J. Performance measurement instrumentation for multiprocessor computers. (appears in) **High Performance Computer Systems**, E. Gelenbe (ed.), Elsevier Science Publishers B.V., 1988, 81-92.

[DON88] Dongarra, J.J. Performance of various computers using standard linear equations software in a FORTRAN environment. Technical Memorandum (issued periodically), Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. 20pp.

[DUB88] Dubois, M., Scheurich, C., and Briggs, F. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Trans. Computers*, (Feb. 1988), 9-21.

[GOT84] Gottlieb, A. and Kruskal, C.P. Complexity results for permuting data and other computations on parallel processors. *Jour. ACM 31*, 2(April, 1984), 193-209.

[GRU86] Grunwald, D.C. and Reed, D.A. Benchmarking hypercube hardware and software. Report No. UIUODCS-R-86-1303 (Nov. 1986), Dept. of Computer Science, Univ. of Ill., 17pp.

[GUS88] Gustafson, J.L. Reevaluating Amdahl's law. *Comm. ACM 31*, 5(May, 1988), 532-533.

[HIL85] Hillis, D. Unpublished remarks on architectural similarity. 12th Annual Int. Symp. on Computer Architecture, Boston, June, 1985.

[HOC84] Hockney, R.W. and Jesshope, C.R. **Parallel Computers.** Adam Hilger Ltd., Bristol, 1984 edition.

[HWA84] Hwang, K. and Briggs, F. **Computer Architecture and Parallel Processing**. New York,NY, McGraw Hill,Inc., 1984.

[IRA88] Irani, K.B., and Luc, K.-Q. Elimination of bottlenecks in dynamic dataflow processors. *Proc., Supercomputing '88*, Kissimmee, FL., November, 1988.

[KUN85] Kung, H.T. Memory requirements for balanced computer architectures. *Journal of Complexity 1*, 1(1985).

[LAK87] Lakshmivarahan, S. *et al.* An experimental study in process communication, the ring benchmark on Multimax - part 1. University of Oklahoma, Norman, Oklahoma. November 1987.

[LAK88] Lakshmivarahan, S. *et al.* An experimental study in process communication, the ring benchmark on Alliant FX/4 - part 2. University of Oklahoma, Norman, Oklahoma. January 1988.

[LAU78] Lauer, H. and Needham, R.M. On the duality of operating systems structures. *Proc., Second International Symposium on Operating Systems Structures*, INRIA (Oct. 1978), (reprinted in *Operating Systems Review* 13, 2 April 1979, 3-19).

[LUB88] Lubeck, O.M. Supercomputer performance: the theory, practice, and results. (appears in) **Advances in Computers, Volume 27**, M.C. Yovits (ed.), Academic Press, Inc, Boston, 1988, 309-362.

[LYO87] Lyon, G.E. On parallel processing benchmarks. NBSIR 87-3580, June, 1987, 35pp.

[LYO88] Lyon, G.E. Design factors for parallel processing benchmarks. (to appear in) *Jour. of Theoretical Computer Science*, (April, 1989).

[LYO88a] Lyon, G.E. Constructing capacity-and-use trees. Parallel Processing Group, NIST, Note, Nov., 1988.

[MOL72] Moler, C. Matrix computations with FORTRAN and paging. *Comm. ACM 15*, 4(April, 1972), 268-270.

[NI87] Ni, L., King, C. and Phillip, P. Parallel algorithm design considerations for hypercube multiprocessors. *Proc. 1987 Int. Conf. on Parallel Processing*, August, 1987, 717-720.

[POT87] Potier, D. Analysis of determinant factors for the performance of vector machines. (appears in) **Supercomputing**, A. Lichnewsky and C. Saquez (eds.), Elsevier Science Publishers B.V. (North-Holland), 1987, 221-236.

[QUI87] Quinn, M.J. **Designing Efficient Algorithms for Parallel Computers.** New York, N.Y., McGraw-Hill,Inc., 1987.

[RED88] Reddaway, S.F. Achieving high performance applications on the DAP. *Proc., CONPAR 88, Stream 'A'* (Brit. Comp. Soc., 1988), 233-241.

[REE88] Reeves, A.P. and Gutierrez, M. On measuring the performance of a massively parallel processor. *Proc., Int. Conf. on Parallel Processing*, Aug. 1988, 261-270.

[SAA88] Saad, Y. and Schultz, M. Topological properties of hypercubes. *IEEE Trans. Computers*, (July, 1988), 867-872.

[WAN88] Wang, J.C., Gary, J.M., and Iyer, H.K. On the analysis of computer performance data. Draft

paper, NIST, Dec. 1988, 33pp.

[WAR72] Ware, W.H. The ultimate computer. *IEEE Spectrum*, (March, 1972),84-91.

[WHI69] White, R.G.S. Rating scale estimates automobile drag coefficient. *SAE Journal 77*, 6(June, 1969), 52-53.

[VAN87] van Waveren, G.M. Application of sparse vector techniques on a molecular dynamics program. (appears in) **Algorithms and Applications on Vector and Parallel Computers**, H.J.J. te Riele, Th.J. Dekker and H.A. van der Vorst (eds.), Elsevier Science Publishers B.V. (North-Holland), 1987, 405-428.

[VAN88] van der Steen, A.J. Proposals for standard benchmark programs for supercomputers. *Proc., CONPAR 88, Stream 'C'* (Brit. Comp. Soc., 1988), 58-66.

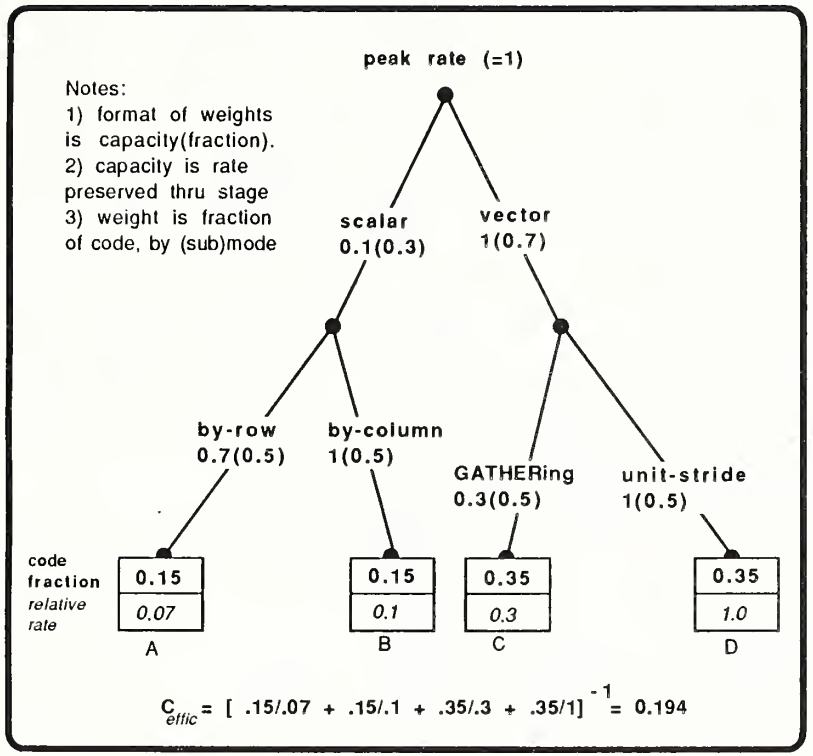


Figure 1. CUT Diagram for Hypothetical System XXX

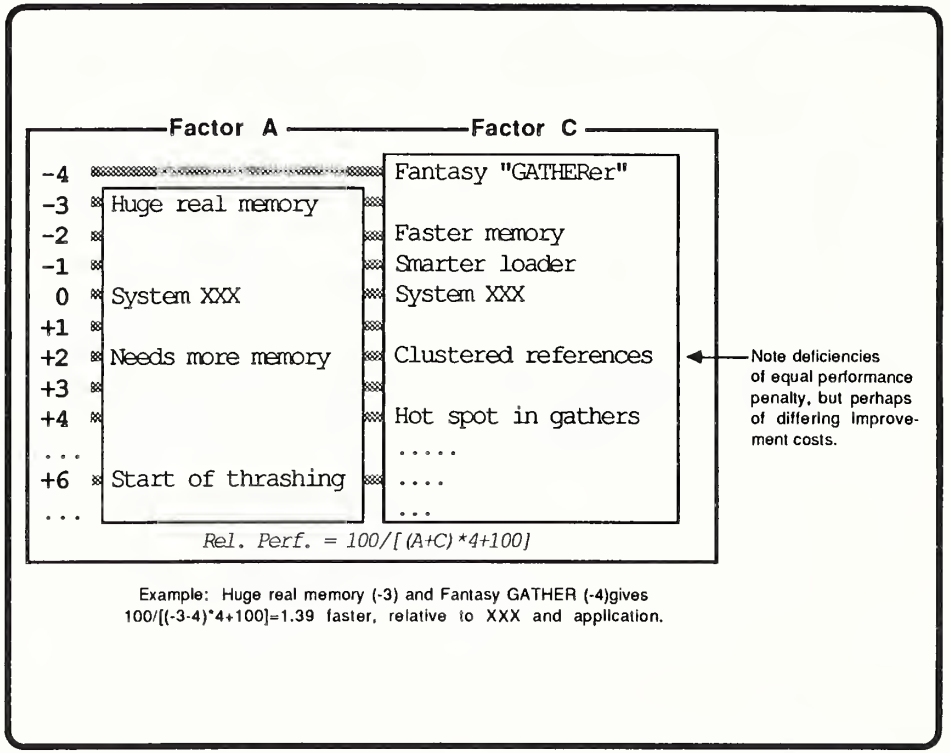


Figure 2. Incremental Performance Changes for Hypothetical System XXX, Factors A and C

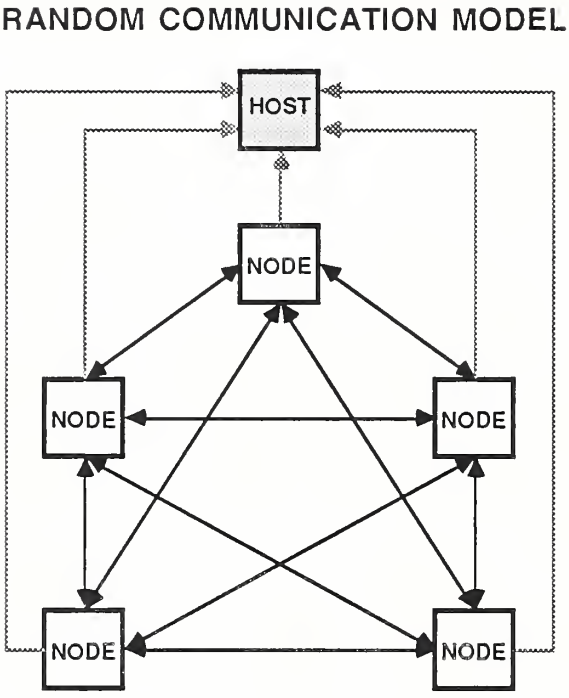


Figure 3a. Communication Patterns, Five Nodes

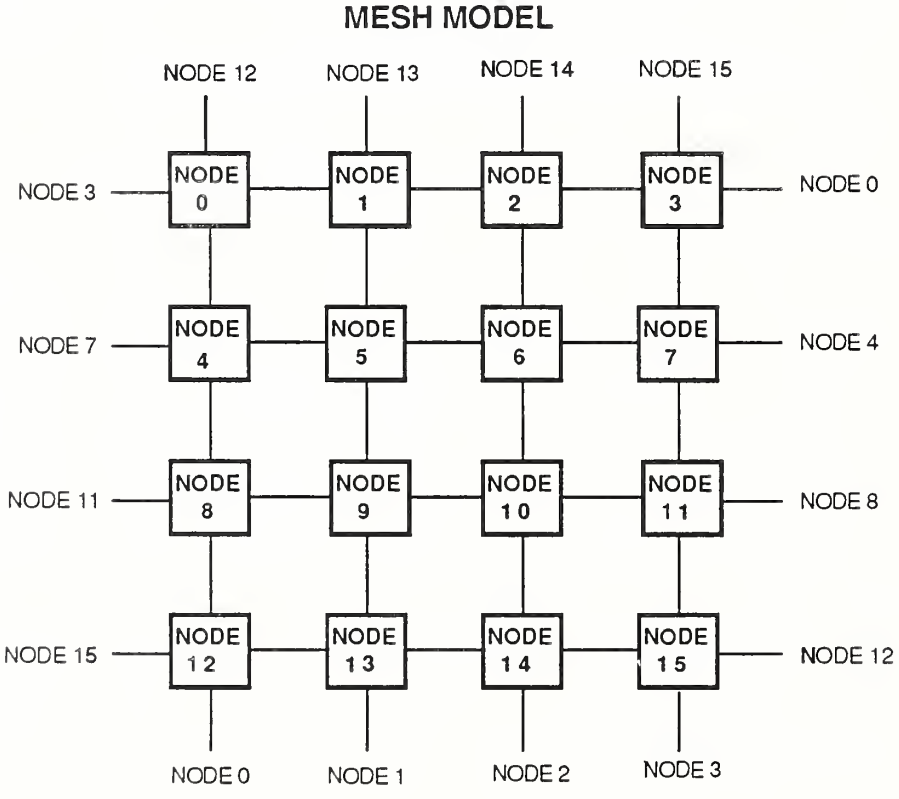


Figure 3b. Communication Patterns, 4x4 Mesh

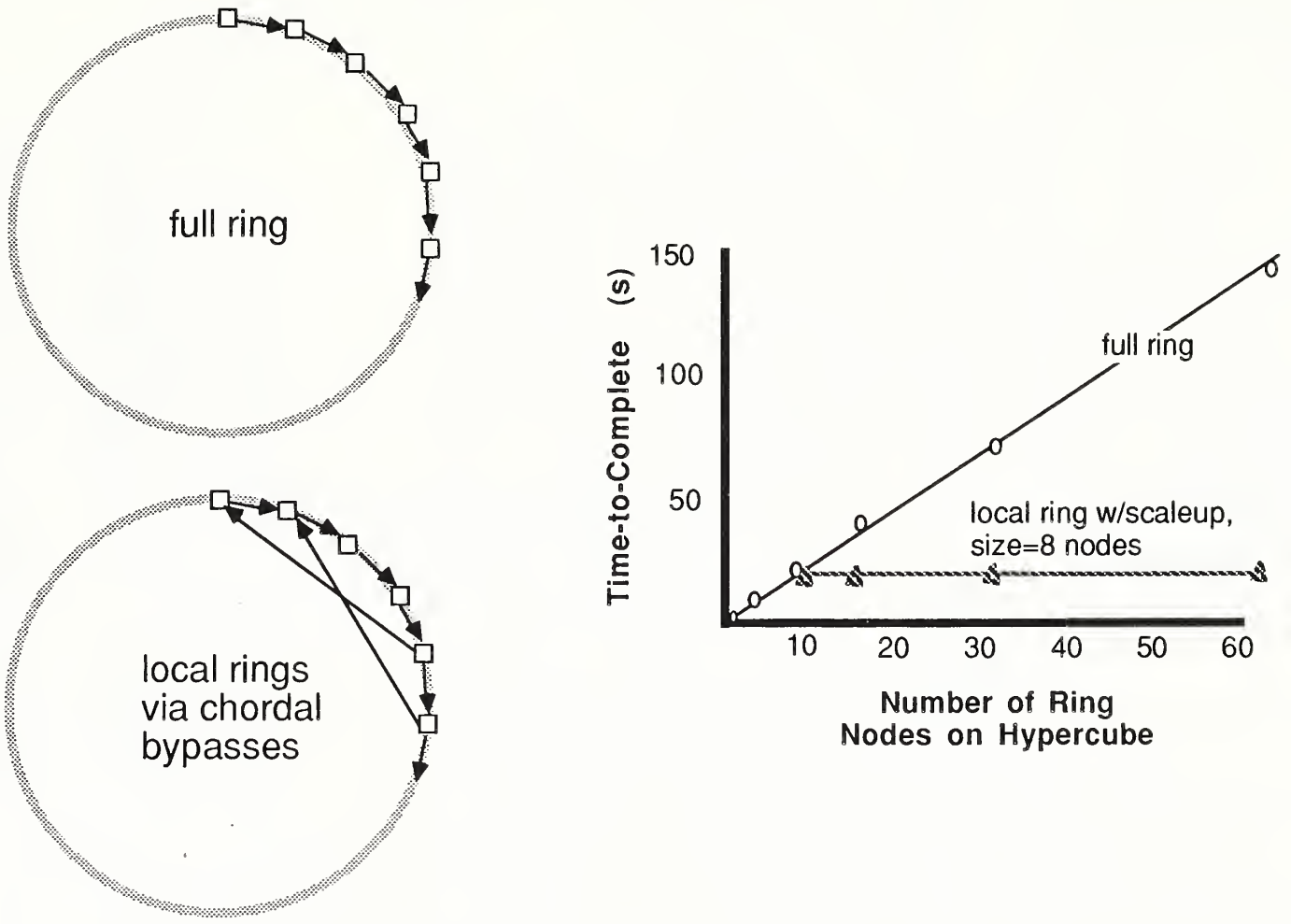
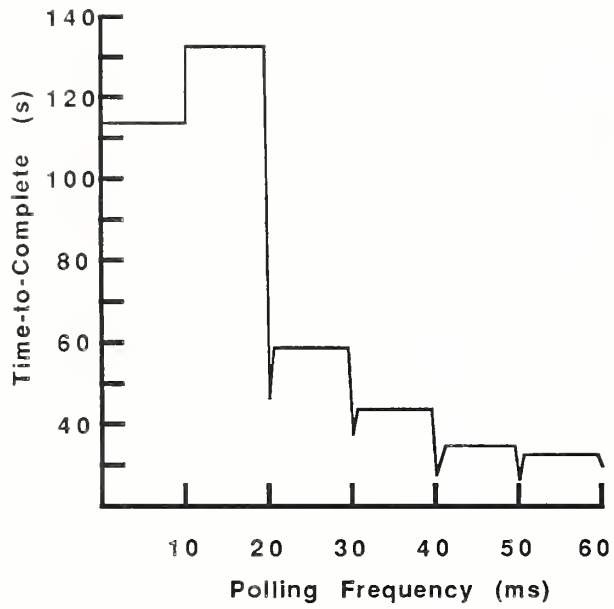
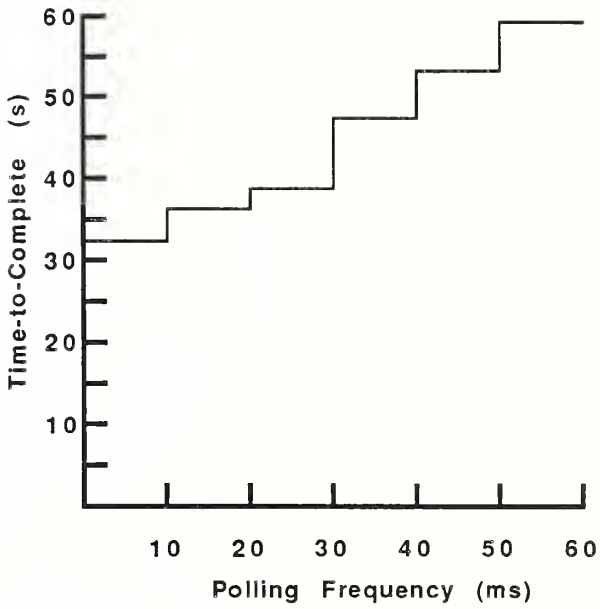


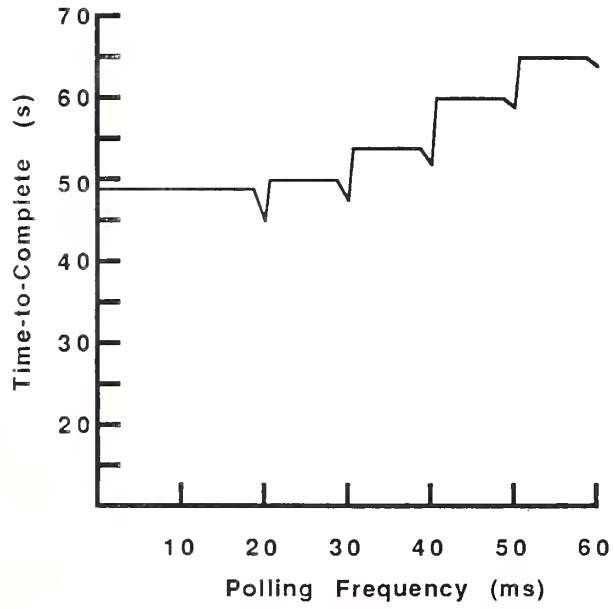
Figure 4. Scaleup: Ring-Pipeline With and Without Local Neighborhoods



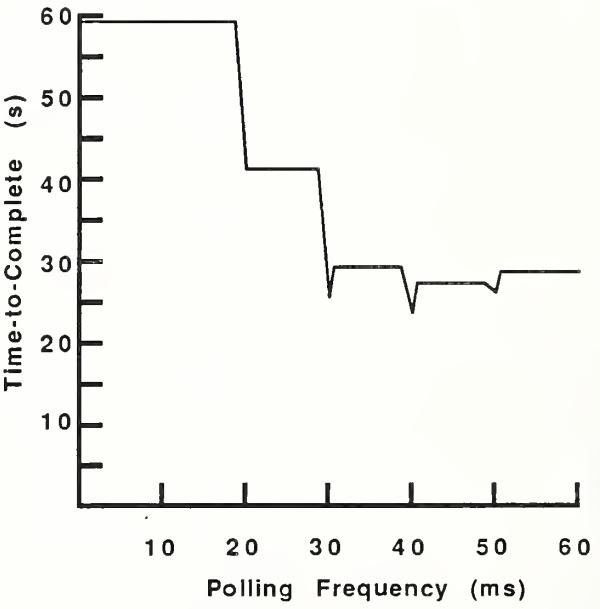
(a) 30 node ring on six processor Balance 8000.



(b) 6 node ring on six processor Balance 8000.



(c) 8 node ring on six processor Balance 8000.



(d) 30 node ring on twenty-four processor Balance 21000

Figure 5. Ring and Polling: Time-to-Complete versus Polling Frequency

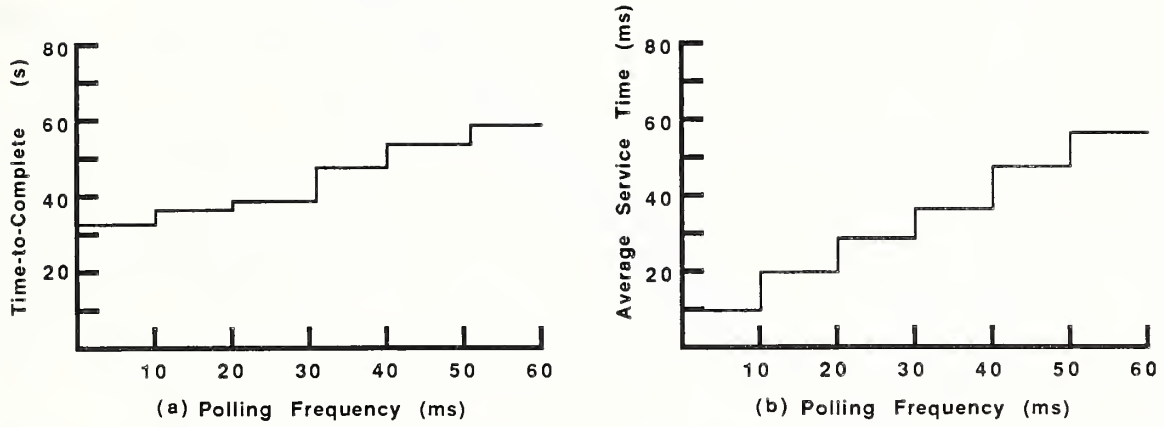


Figure 6. 6-Node Ring on Six Processor Sequent Balance 8000

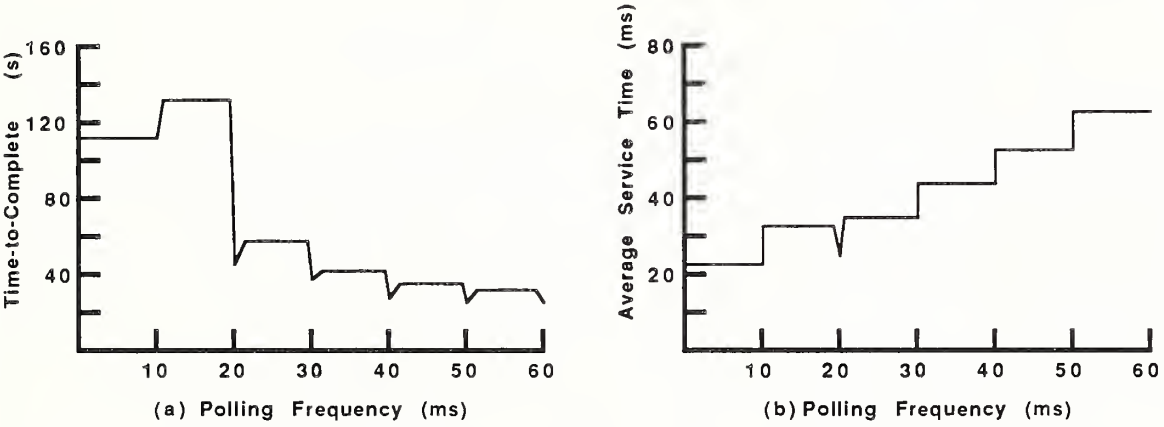


Figure 7. 30-Node Ring on Six Processor Sequent Balance 8000

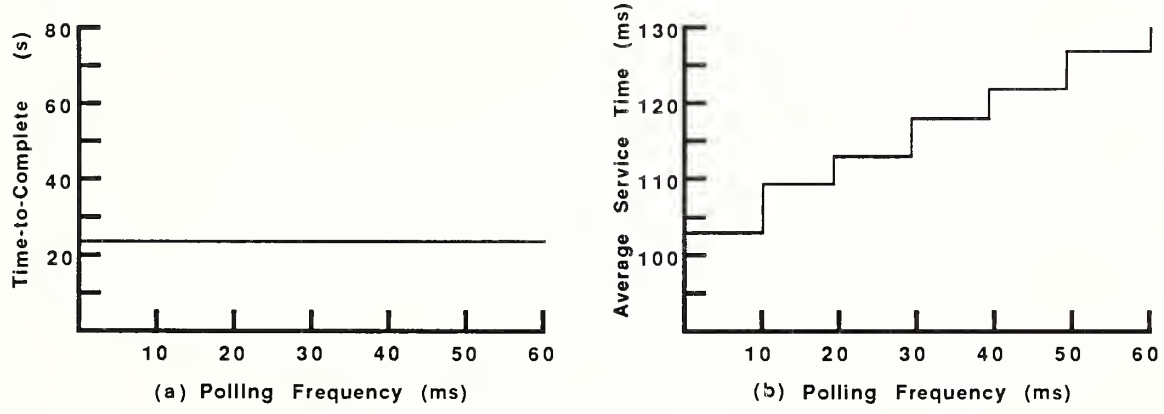


Figure 8. 30-Node Ring (Bound Processes) on Six Processor Sequent Balance 8000

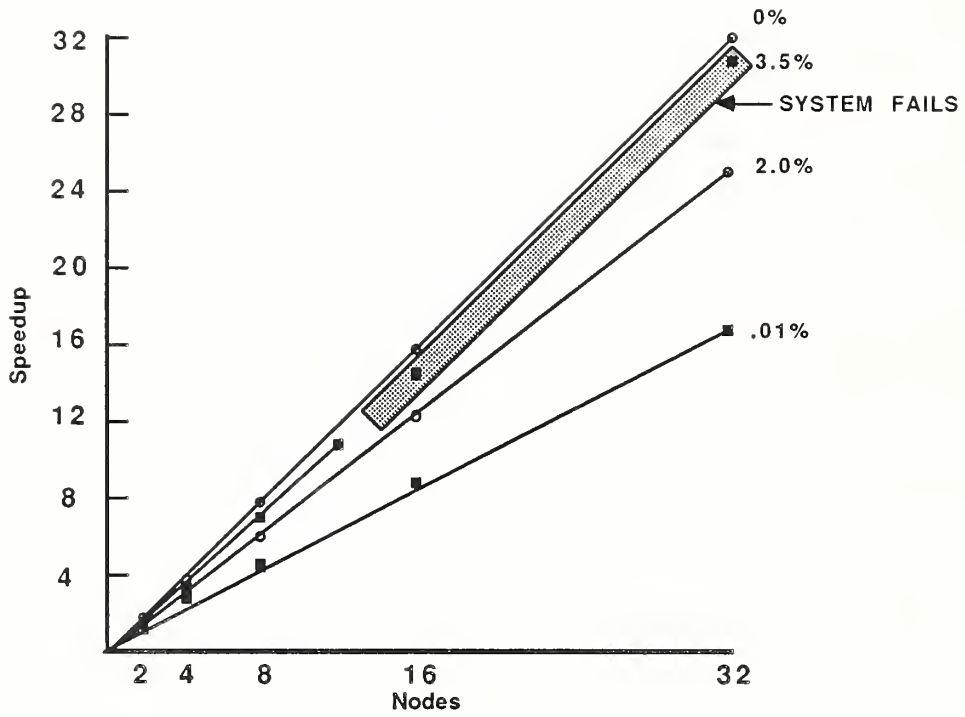


Figure 9. Speedup for Random Communication with Various Communication Frequencies (Hypercube)

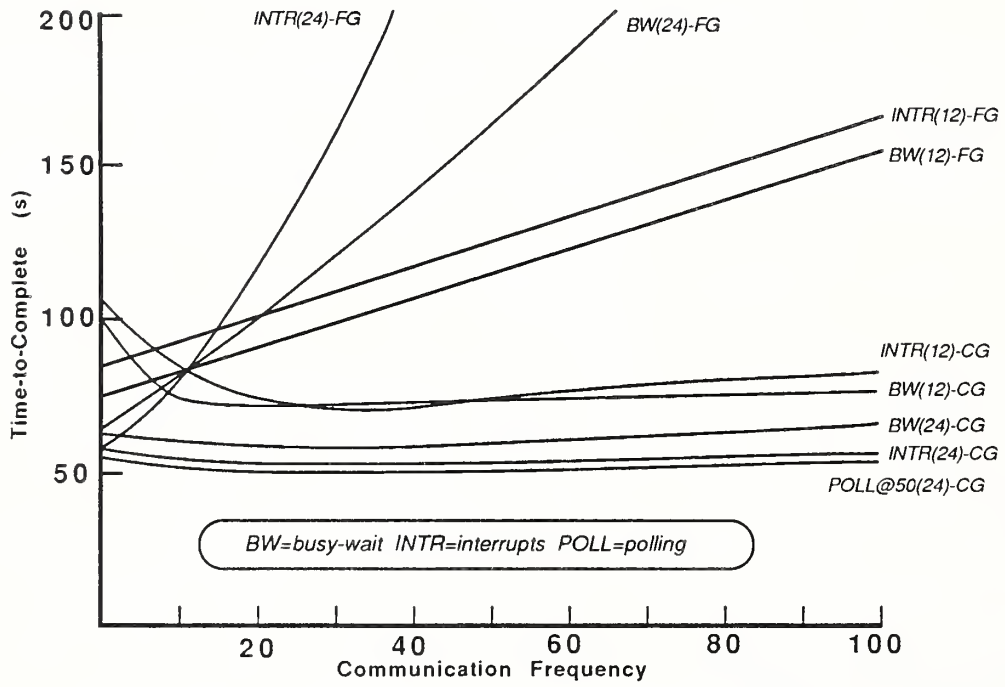


Figure 10. Time, Frequencies and Synchronization Strategies (16 Processor, Shared Memory)

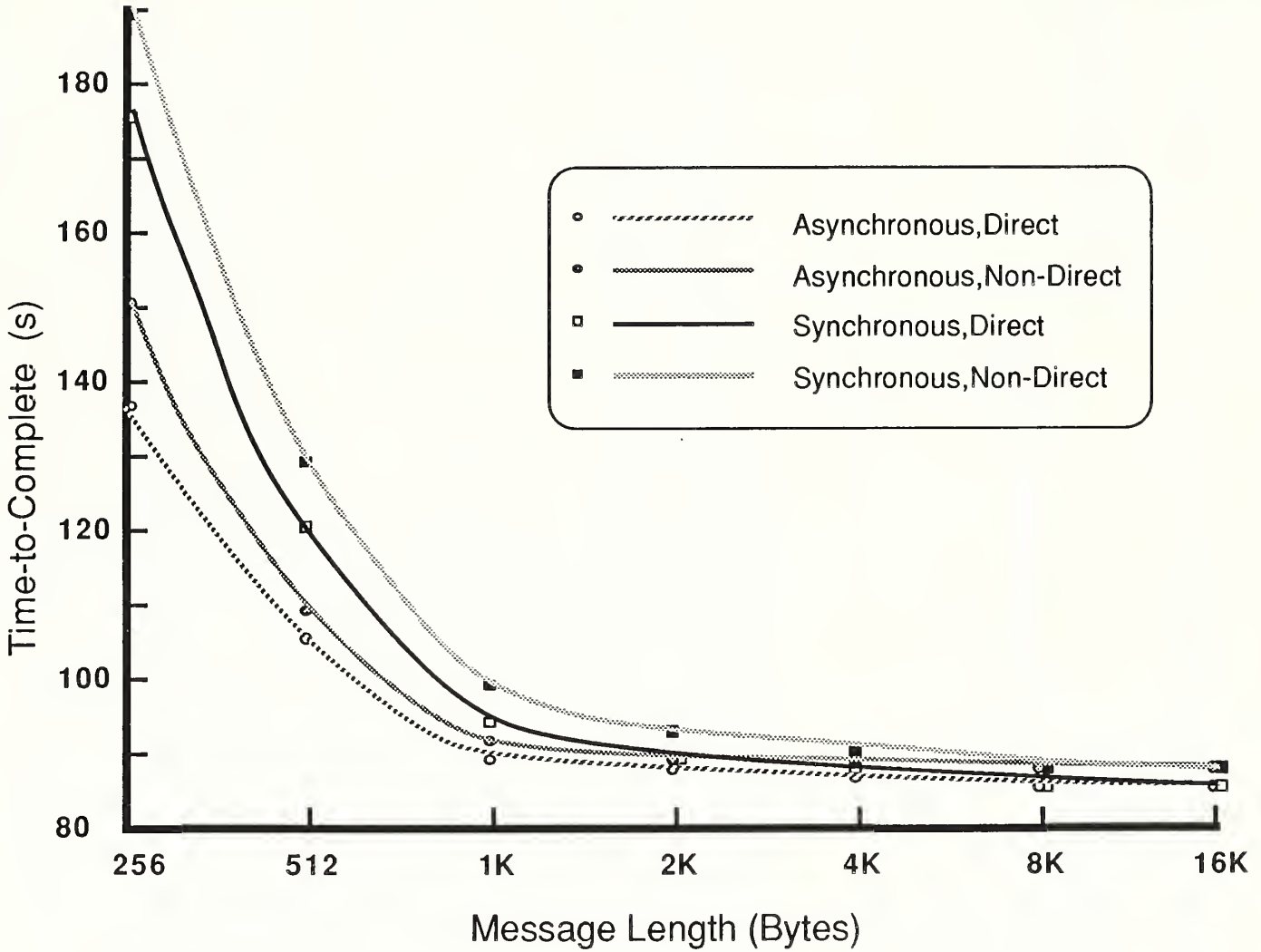


Figure 11. Time versus Message Length, 4x4 Mesh with Various Synchronization and Mapping Strategies (Hypercube)

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NISTIR 89-4053	2. Performing Organ. Report No.	3. Publication Date MARCH 1989

4. TITLE AND SUBTITLE

Architecturally-Focused Benchmarks with a Communication Example

5. AUTHOR(S)

G. E. Lyon and R. D. Snelick

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The discussion first sketches a framework of modalities for an architecturally-focused performance evaluation. The result is a hybrid of benchmarking and modeling: Elements of capacity-and-use trees, CUTs, are explored as a simplified notation. There follows a description of the structure and preliminary results from a practical benchmark set for process communication.

Argument is given that performance within a class of architecture is often dominated by unavoidable competitions within distinct machine modalities, such as scalar-vector. A k-alternative, forced choice defines a dimension of comparison equally well in SI- or MIMD architectures. Performance estimators are interpolations between values from basis benchmarks for modes; ideally in the two-alternative forced choice only two benchmark measurements are needed. Refinements in basis benchmarks support CUT-based estimates of performance.

The example set of communication benchmarks shows how refinements can clarify knowledge of a machine. The refining expands details for a given mode. Ring, mesh, and random connection benchmarks demonstrate diagnostic details on a particular mode (process communication) on a machine. The results sample both shared-memory and message-passing, and cover architecture influence, synchronization mechanisms, message, computational and synchronization granularities, and mappings of logical to physical structures. Emphasis is upon capturing important characteristics and capabilities of a machine's communication subsystem.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

architecture; benchmarks; measurements; metrics; models; performance; synthetics

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

38

15. Price

$12.95